Advanced Lane Finding Project

Advanced Lane Finding Project

The goals / steps of this project are the following:

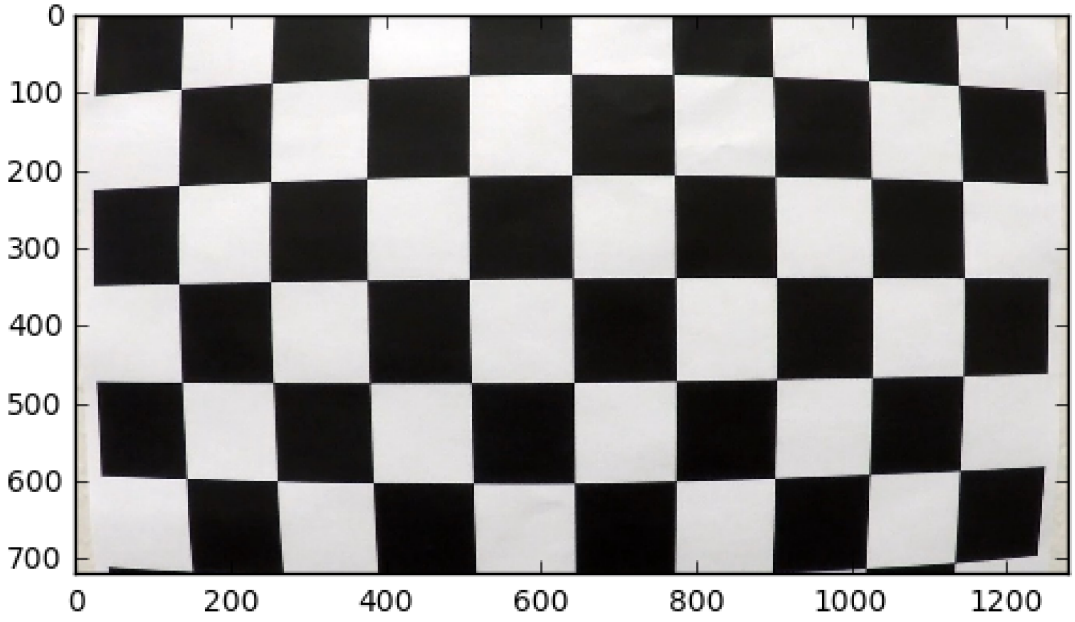
- Compute the camera calibration matrix and distortion coefficients given a set of chessboard images.
- Apply a distortion correction to raw images.
- Use color transforms, gradients, etc., to create a thresholded binary image.
- Apply a perspective transform to rectify binary image ("birds-eye view").
- Detect lane pixels and fit to find the lane boundary.
- Determine the curvature of the lane and vehicle position with respect to center.
- Warp the detected lane boundaries back onto the original image.
- Output visual display of the lane boundaries and numerical estimation of lane curvature and vehicle position.

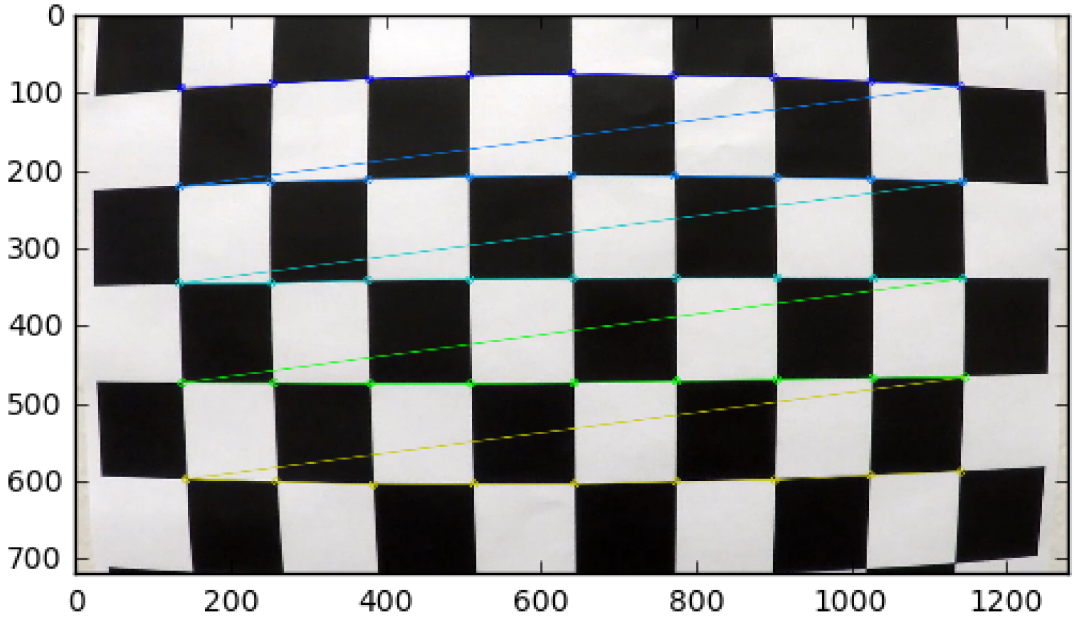
Camera Calibration

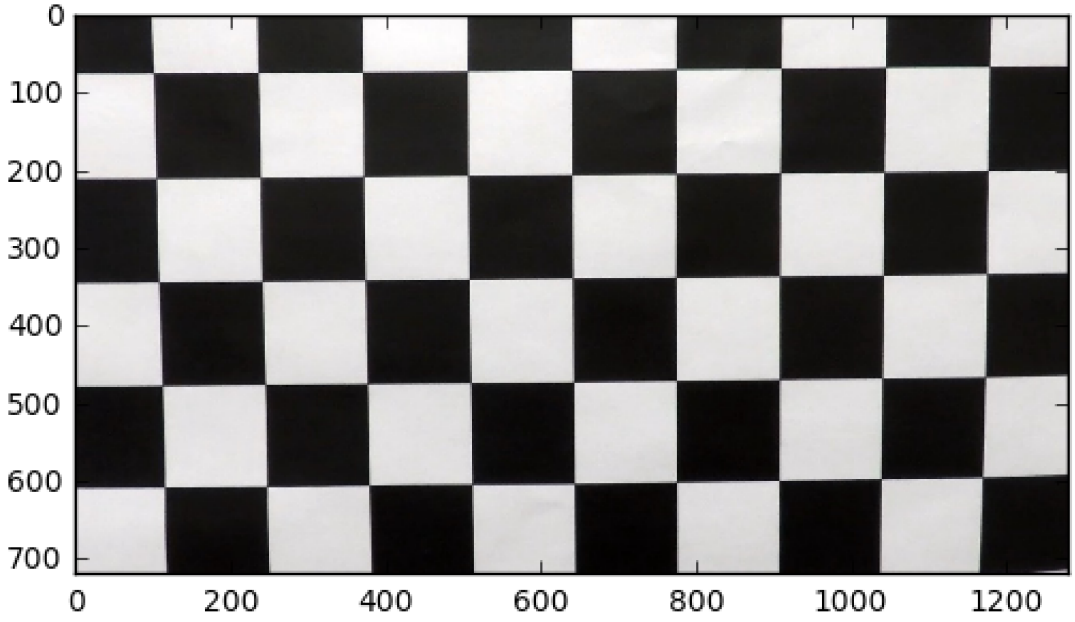
The code for this step is contained in the fourth code cell of the IPython notebook which is submitted with this project.

I basically followed the steps illustrated in the course. The camera calibration was described very well and I start by preparing "object points", which will be the (x, y, z) coordinates of the chessboard corners in the world. Here I am assuming the chessboard is fixed on the (x, y) plane at z=0, such that the object points are the same for each calibration image. Thus, `objp` is just a replicated array of coordinates, and `objpoints` will be appended with a copy of it every time I successfully detect all chessboard corners in a test image. `imgpoints` will be appended with the (x, y) pixel position of each of the corners in the image plane with each successful chessboard detection.

I then used the output `objpoints` and `imgpoints` to compute the camera calibration and distortion coefficients using the `cv2.calibrateCamera()` function. I applied this distortion correction to the test image using the `cv2.undistort()` function and obtained these results, the first image is one of the distorted images given for camera calibration, the second image is the results of corner detection on the first image and the third image is the result if distortion correction on the first image:


0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200


0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200



Pipeline (single images)

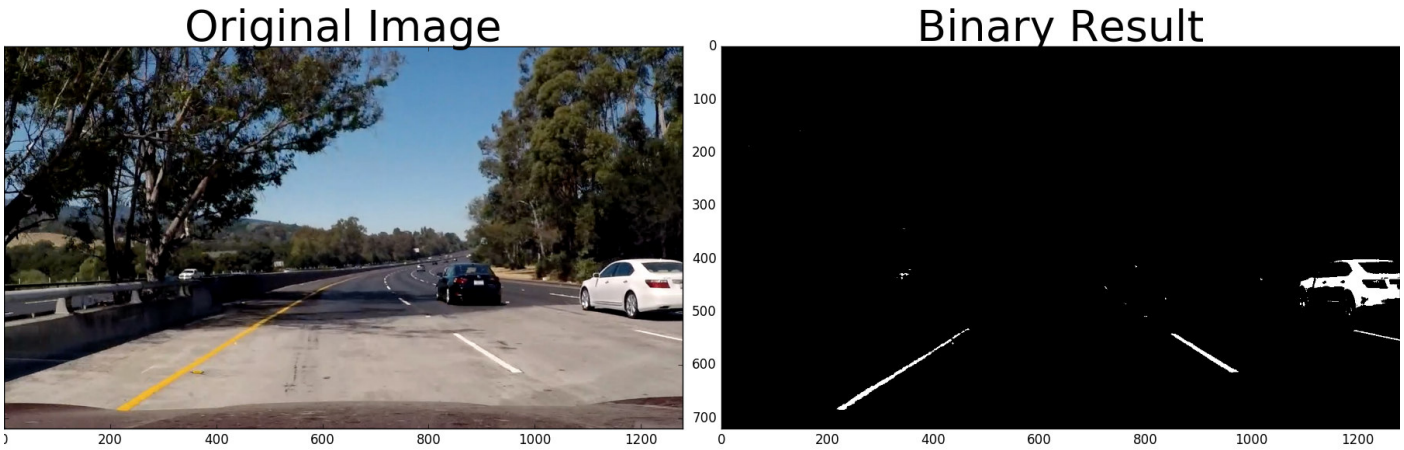
1. Provide an example of a distortion-corrected image.

To demonstrate this step, I will describe how I apply the distortion correction to one of the test images like this one:

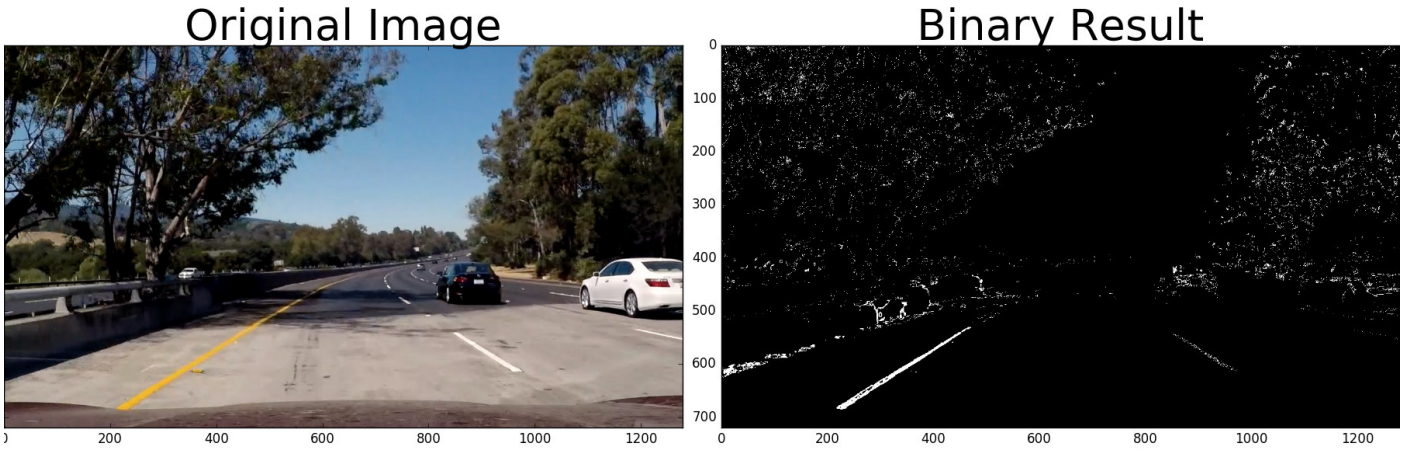
2. Gradient image.

A good portion of my time was spent to study and experiment different gradient methods we learn in the

course to create a binary image. At the end I used a combination of three thresholded image to create my gradient image. First, I used the r-channel of the original RGB image to utilize the color information in the image. The value is used to threshold the r-channel image was (230, 255), and that value resulted in an image below:



Then I used the HSV color conversion to convert the image to HSV space and I used the s-channel, which has the least effect of the light variation, to create a second binary image. The threshold value for this image was (180, 230). The resulted image is shown below:



And finally I used the gradient image in x-axis to find pattern in horizontal axis. The threshold value for the x-axis Sobel operator was (50,100). And then I combined all three binary images created in the above procedures. The result is shown below:

This image had pretty much all the binary information required to find the lanes. The gradient function is in the seventh cell of the python notebook.

3. Perspective Transform

Perspective transform is a separate process that could sit outside of the pipeline. It is a one time procedure that results in the perspective transformation and the transformation will be used to convert each video frame to birds eye view.

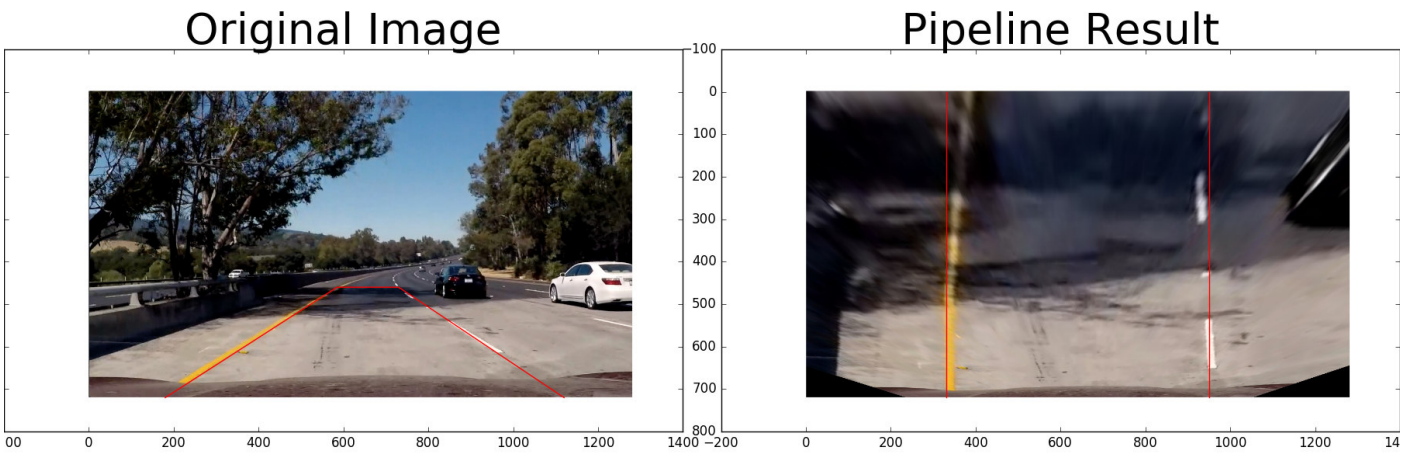
The code for my perspective transform is in the third cell of my python notebook. The function uses the hardcoded values of the source and destination points in the following manner:

``` src points: lb = [180, image.shape[0]] rb = [image.shape[1]-160, image.shape[0]] lu = [585, image.shape[0]/2+100] ru = [730, image.shape[0]/2+100] dst = np.float32( [[(img*size[0] / 4), 0], [(img*size[0] / 4), img*size[1]], [(img*size[0] * 3 / 4), img*size[1]], [(img*size[0] * 3 / 4), 0]])

``` This resulted in the following source and destination points:

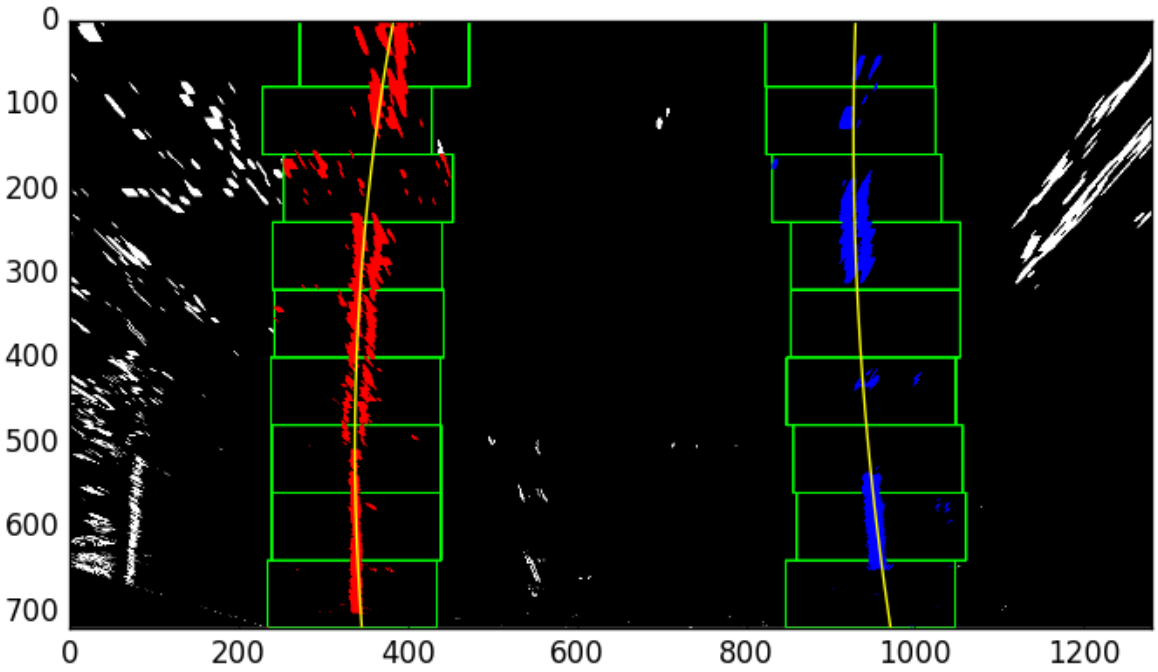
| Source | Destination | |:-------------:|:-------------:| | 180, 720 | 320, 0 | | 1040, 720 | 320, 720 | | 585, 460 | 960, 720 | | 730, 460 | 960, 0 |

I verified that my perspective transform was working as expected by drawing the `src` and `dst` points onto a test image and its warped counterpart to verify that the lines appear parallel in the warped image.

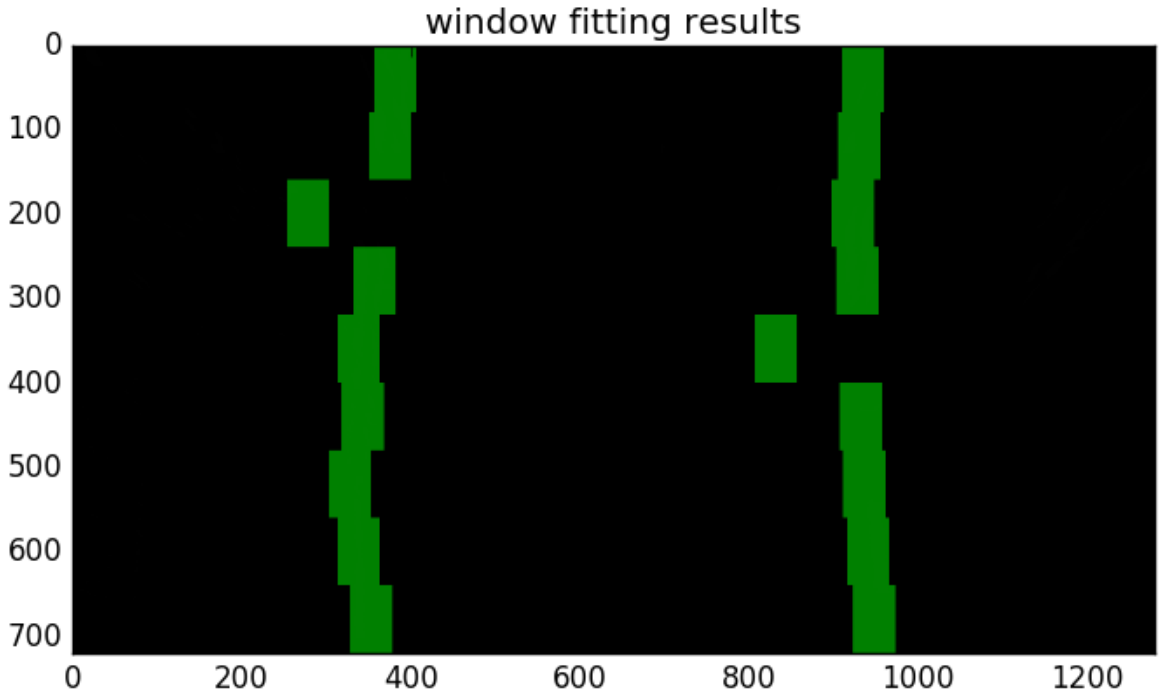



## 4. Lane-line identification and polynomial fit

I used the sliding window algorithm to find the lane-line pixels and the second order polynomial, as described in the course. I implemented the method to find the next frame lane-line using the starting point from the previous frame, so it was not required to do an initial search for each frame.




I also implemented the convolution method to find the lane-lines, however the result did not show the same accuracy as the sliding window, so I decided to use the sliding window. The result of convolution is shown below:




The code for this part of the project is in the 8th and 9th cells of the python notebook.

### 5. Lane curvature radius and Center lane Position.

Lane curvature computation has two components in it, first we have to project the left and right line pixels into the world coordinate system. This is done by using a conversion from pixel to meter space. The second component is to find a polynomial and use the derivative of the polynomial as described in the course material to find the lane curvature.

For the lane center offset, I used the mean position of the all x pixels of the left and right lines. Since the center of the image in x-axis is the half of the x-dimension of the image, the offset can be computing by subtracting the position of the above values.

The code for computing the lane curvature and lane offset is in the 10th cell of the Pyhton notebook.

### 6. Sanity Check

For sanity check I used all the three suggested methods. First I used a check to see if the left and right lines are parallel. To do so, I used the dot product of the left and right lines and computed the cosine of the angles between the lines. The cosine value shows how parallel these lines are. I set the threshold

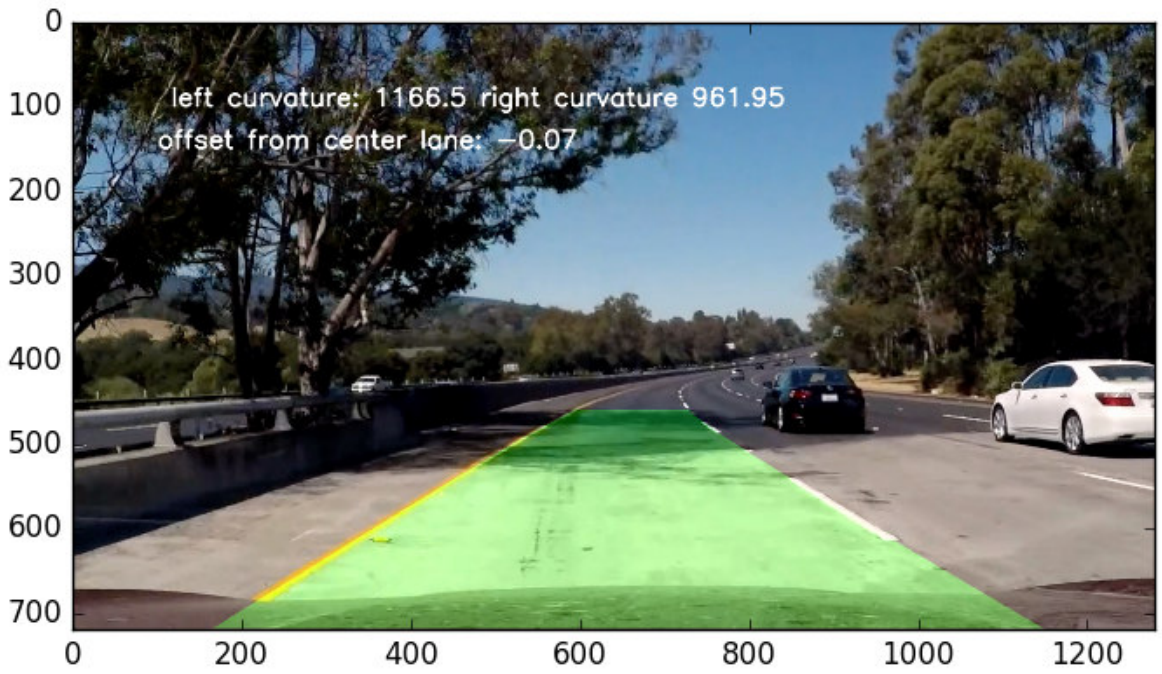
for how parallel the lines are to 0.9.

Then I used the distance between center of the lane and the offset value resulted from lane-line identification and if this distance was more than 100 pixels, I failed the sanity check.

And as the third measure, if the difference between lane curvatures for left and right was more than 50 meter, I again failed the sanity check. A failed sanity check meant that it is required to find the lanes using the sliding window and the information from the previous frame could not be used. The code for sanity check is in the 12th cell of the pipeline.

### 6. Pipeline Result of a Single Image

The above steps were applied to a single image and result is shown as below:




## Pipeline (video)

The code to process the project video is in the last cell of my python notebook. Each frame of the vidoe is passed to the complete pipeline and the result is saved in the output video.

Here's a link to my video result

## Discussion

A few things that I had to fine tune in order to get the pipeline working. I had to change the margin of the next frame processing method to avoid the lane identification to jump into the side curb of the bridges. The value I chose for margin was 50 pixel, instead of the 100, to have the search for the next frame starting pixels closer to the current lines.

Some of the challenges of the project included; the processing time for each frame was more than optimal, meaning that the pipeline in its current format cannot be used in realtime. It took around 3-4 minutes to process the whole project video (54 seconds).

When I applied the pipeline to the challenge video, the result was not acceptable. Trying to tune the parameters of the current pipeline was not successful. I believe more image processing techniques are required, and just trying to find a better threshold, or changing the sliding window, etc. will not be enough to correctly identify lane lines in the challenge video.